PATENT APPLICATION

PROCESSING ARCHITECTURE HAVING A COMPARE CAPABILITY

Inventor(s):

Ashley Saulsbury, a citizen of Great Britain, residing at,
18488 Grizzly Rock Rd.
Los Gatos, CA 95033

Assignee:

Sun Microsystems, Inc
901 San Antonio Road
Palo Alto, CA 94303

Entity: Large

PROCESSING ARCHITECTURE HAVING A FLOATING POINT COMPARE CAPABILITY

This application claims the benefit of U.S. Provisional Application No. 60/187,651 filed on March 8, 2000.

5   CROSS-REFERENCES TO RELATED APPLICATIONS

This application is being filed concurrently with related U.S. patent applications: Attorney Docket Number 016747-00991, entitled "VLIW Computer Processing Architecture with On-chip DRAM Usable as Physical Memory or Cache Memory"; Attorney Docket Number 016747-01001, entitled "VLIW Computer

10   Processing Architecture Having a Scalable Number of Register Files"; Attorney Docket Number 016747-01780, entitled "Computer Processing Architecture Having a Scalable Number of Processing Paths and Pipelines"; Attorney Docket Number 016747-01051, entitled "VLIW Computer Processing Architecture with On-chip Dynamic RAM"; Attorney Docket Number 016747-01211, entitled "Computer Processing Architecture

15   Having the Program Counter Stored in a Register File Register"; Attorney Docket Number 016747-01461, entitled "Processing Architecture Having Parallel Arithmetic Capability"; Attorney Docket Number 016747-01471, entitled "Processing Architecture Having an Array Bounds Check Capability"; Attorney Docket Number 016747-01481, entitled "Processing Architecture Having an Array Bounds Check Capability"; and,

20   Attorney Docket Number 016747-01521, entitled "Processing Architecture Having a Matrix Transpose Capability"; all of which are incorporated herein by reference.

        BACKGROUND OF THE INVENTION

The present invention relates generally to an improved computer

25   processing instruction set, and more particularly to an instruction set having a compare functionality.

Computer architecture designers are constantly trying to increase the speed and efficiency of computer processors. For example, computer architecture designers have attempted to increase processing speeds by increasing clock speeds and attempting

30   latency hiding techniques, such as data prefetching and cache memories. In addition, other techniques, such as instruction-level parallelism using VLIW, multiple-issue superscalar, speculative execution, scoreboarding, and pipelining are used to further

enhance performance and increase the number of instructions issued per clock cycle (IPC).

Architectures that attain their performance through instruction-level parallelism seem to be the growing trend in the computer architecture field. Examples of architectures utilizing instruction-level parallelism include single instruction multiple data (SIMD) architecture, multiple instruction multiple data (MIMD) architecture, vector or array processing, and very long instruction word (VLIW) techniques. Of these, VLIW appears to be the most suitable for general-purpose computing.

Instruction sets typically include one or more compare instructions. These instructions compares two input registers so that decisions can be made based upon the result. For example, a branch in software is a compare function followed by checking the result to decide which branch to follow. The output from the compare instruction travels through a dedicated path to a dedicated output register. However, use of a single dedicated register is undesirable in VLIW systems because two separate data paths may want to access the dedicated register at the same time. To accommodate a single dedicated output register, a VLIW system would schedule compare instructions so that none overlapped. Additionally, use of a dedicated path to the output register consumes die area and increases the circuit complexity.

Compare instructions only determine a single relationship between two numbers. For example, a compare instruction answers the question of whether one number is either less than, greater than or equal to another number with a response of true or false. However, separate compare instructions are needed if answers to multiple compare-type questions are needed. As the compare instructions needed for answering multiple questions are executed sequentially, this extends the time needed to answer a multitude of compare-type questions.

SUMMARY OF THE INVENTION

The present invention compares source operands and stores a result in a register where the result indicates a plurality of mathematical relationships between the source operands. In one embodiment, a processing core executes a compare instruction. The processing core includes a register file, comparison logic, decode logic, and a store path. Included in the register file are a number of general-purpose registers. The general-purpose registers include a first input operand register, a second input operand register and an output operand register. Comparison logic is coupled to the register file. The

2

comparison logic tests for at least two of the following relationships: less than, equal to, greater than and no valid relationship. The decode logic selects the output operand register from the plurality of general-purpose registers. The store path extends between the comparison logic and the selected output operand register.

5   A more complete understanding of the present invention may be derived by referring to the detailed description of preferred embodiments and claims when considered in connection with the figures, wherein like reference numbers refer to similar items throughout the figures.

10   BRIEF DESCRIPTION OF THE DRAWINGS
          Fig. 1 is a block diagram of an embodiment of a processor chip having the processor logic and memory on the same integrated circuit;
          Fig. 2 is block diagram illustrating one embodiment of a processing core having a four-way VLIW pipeline design;
15   Fig. 3 is a diagram showing some data types generally available to the processor chip;
          Fig. 4 is a diagram showing an embodiment of machine code syntax for the compare sub-instruction in one embodiment;
          Fig. 5 is a block diagram which schematically illustrates an embodiment of
20   the compare function; and
          Fig. 6 is a flow diagram of an embodiment of a method for comparing two operands and storing the result in a general-purpose register.

                     DESCRIPTION OF THE SPECIFIC EMBODIMENTS

25   Introduction
          The present invention provides a novel computer processor chip having a sub-instruction for comparing registers which performs multiple comparisons on input registers. The result from the comparison is stored in a general-purpose register in the register file. As one skilled in the art will appreciate, performing multiple compares in a
30   single instruction issue increases efficiency. Additionally, storing the result in general-purpose register file improves efficiency in architectures which issues compares either simultaneously or in quick succession to each other. The result indicates a number of

3

mathematical relationships between the source operands without issuing separate instructions.

In the Figures, similar components and/or features have the same reference label. Further, various components of the same type are distinguished by following the reference label by a dash and a second label that distinguishes among the similar components. If only the first reference label is used in the specification, the description is applicable to any one of the similar components having the second label.

Processor Overview

With reference to Fig. 1, a processor chip 10 is shown which embodies the present invention. In particular, processor chip 10 comprises a processing core 12, a plurality of memory banks 14, a memory controller 20, a distributed shared memory controller 22, an external memory interface 24, a high-speed I/O link 26, a boot interface 28, and a diagnostic interface 30.

As discussed in more detail below, processing core 12 comprises a scalable VLIW processing core, which may be configured as a single processing pipeline or as multiple processing pipelines. The number of processing pipelines typically is a function of the processing power needed for the particular application. For example, a processor for a personal workstation typically will require fewer pipelines than are required in a supercomputing system.

In addition to processing core 12, processor chip 10 comprises one or more banks of memory 14. As illustrated in Fig. 1, any number of banks of memory can be placed on processor chip 10. As one skilled in the art will appreciate, the amount of memory 14 configured on chip 10 is limited by current silicon processing technology. As transistor and line geometries decrease, the total amount of memory that can be placed on a processor chip 10 will increase.

Connected between processing core 12 and memory 14 is a memory controller 20. Memory controller 20 communicates with processing core 12 and memory 14, and handles the memory I/O requests to memory 14 from processing core 12 and from other processors and I/O devices. Connected to memory controller 20 is a distributed shared memory (DSM) controller 22, which controls and routes I/O requests and data messages from processing core 12 to off-chip devices, such as other processor chips and/or I/O peripheral devices. In addition, as discussed in more detail below, DSM controller 22 is configured to receive I/O requests and data messages from off-chip

4

devices, and route the requests and messages to memory controller 20 for access to memory 14 or processing core 12.

High-speed I/O link 26 is connected to the DSM controller 22. In accordance with this aspect of the present invention, DSM controller 22 communicates

5   with other processor chips and I/O peripheral devices across the I/O link 26. For example, DSM controller 22 sends I/O requests and data messages to other devices via I/O link 26. Similarly, DSM controller 22 receives I/O requests from other devices via the link.

Processor chip 10 further comprises an external memory interface 24.

10   External memory interface 24 is connected to memory controller 20 and is configured to communicate memory I/O requests from memory controller 20 to external memory. Finally, as mentioned briefly above, processor chip 10 further comprises a boot interface 28 and a diagnostic interface 30. Boot interface 28 is connected to processing core 12 and is configured to receive a bootstrap program for cold booting processing core 12

15   when needed. Similarly, diagnostic interface 30 also is connected to processing core 12 and configured to provide external access to the processing core for diagnostic purposes.

Processing Core

20   1.   GENERAL CONFIGURATION

As mentioned briefly above, processing core 12 comprises a scalable VLIW processing core, which may be configured as a single processing pipeline or as multiple processing pipelines. A single processing pipeline can function as a single pipeline processing one instruction at a time, or as a single VLIW pipeline processing

25   multiple sub-instructions in a single VLIW instruction word. Similarly, a multi-pipeline processing core can function as multiple autonomous processing cores. This enables an operating system to dynamically choose between a synchronized VLIW operation or a parallel multi-threaded paradigm. In multi-threaded mode, the VLIW processor manages a number of strands executed in parallel.

30   In accordance with one embodiment of the present invention, when processing core 12 is operating in the synchronized VLIW operation mode, an application program compiler typically creates a VLIW instruction word comprising a plurality of sub-instructions appended together, which are then processed in parallel by processing core 12. The number of sub-instructions in the VLIW instruction word matches the total

5

number of available processing paths in the processing core pipeline. Thus, each processing path processes VLIW sub-instructions so that all the sub-instructions are processed in parallel. In accordance with this particular aspect of the present invention, the sub-instructions in a VLIW instruction word issue together in this embodiment. Thus,

5   if one of the processing paths is stalled, all the sub-instructions will stall until all of the processing paths clear. Then, all the sub-instructions in the VLIW instruction word will issue at the same time. As one skilled in the art will appreciate, even though the sub-instructions issue simultaneously, the processing of each sub-instruction may complete at different times or clock cycles, because different sub-instruction types may have different

10   processing latencies.

In accordance with an alternative embodiment of the present invention, when the multi-pipelined processing core is operating in the parallel multi-threaded mode, the program sub-instructions are not necessarily tied together in a VLIW instruction word. Thus, as instructions are retrieved from an instruction cache, the operating system

15   determines which pipeline is to process each sub-instruction for a strand. Thus, with this particular configuration, each pipeline can act as an independent processor, processing a strand independent of strands in the other pipelines. In addition, in accordance with one embodiment of the present invention, by using the multi-threaded mode, the same program sub-instructions can be processed simultaneously by two separate pipelines

20   using two separate blocks of data, thus achieving a fault tolerant processing core. The remainder of the discussion herein will be directed to a synchronized VLIW operation mode. However, the present invention is not limited to this particular configuration.

2.   VERY LONG INSTRUCTION WORD (VLIW)

25   Referring now to Fig. 2, a simple block diagram of a VLIW processing core pipeline 50 having four processing paths, 56-1 to 56-4, is shown. In accordance with the illustrated embodiment, a VLIW 52 comprises four RISC-like sub-instructions, 54-1, 54-2, 54-3, and 54-4, appended together into a single instruction word. For example, an instruction word of one hundred and twenty-eight bits is divided into four thirty-two bit

30   sub-instructions. The number of VLIW sub-instructions 54 correspond to the number of processing paths 56 in processing core pipeline 50. Accordingly, while the illustrated embodiment shows four sub-instructions 54 and four processing paths 56, one skilled in the art will appreciate that the pipeline 50 may comprise any number of sub-instructions

6

54 and processing paths 56. Typically, however, the number of sub-instructions 54 and processing paths 56 is a power of two.

Each sub-instruction 54 in this embodiment corresponds directly with a specific processing path 56 within the pipeline 50. Each of the sub-instructions 54 are of similar format and operate on one or more related register files 60. For example, processing core pipeline 50 may be configured so that all four sub-instructions 54 access the same register file, or processing core pipeline 50 may be configured to have multiple register files 60. In accordance with the illustrated embodiment of the present invention, sub-instructions 54-1 and 54-2 access register file 60-1, and sub-instructions 54-3 and 54-4 access register file 60-2. As those skilled in the art can appreciate, such a configuration can help improve performance of the processing core.

As illustrated in Fig. 2, an instruction decode and issue logic stage 58 of the processing core pipeline 50 receives VLIW instruction word 52 and decodes and issues the sub-instructions 54 to the appropriate processing paths 56. Each sub-instruction 54 then passes to the execute stage of pipeline 50 which includes a functional or execute unit 62 for each processing path 56. Each functional or execute unit 62 may comprise an integer processing unit 64, a load/store processing unit 66, a floating point processing unit 68, or a combination of any or all of the above. For example, in accordance with the particular embodiment illustrated in Fig. 2, the execute unit 62-1 includes an integer processing unit 64-1 and a floating point processing unit 68; the execute unit 62-2 includes an integer processing unit 64-2 and a load/store processing unit 66-1; the execute unit 62-3 includes an integer processing unit 64-3 and a load/store unit 66-2; and the execute unit 62-4 includes only an integer unit 64-4.

As one skilled in the art will appreciate, scheduling of sub-instructions within a VLIW instruction word 52 and scheduling the order of VLIW instruction words within a program is important so as to avoid unnecessary latency problems, such as load, store and writeback dependencies. In accordance with the one embodiment of the present invention, the scheduling responsibilities are primarily relegated to the software compiler for the application programs. Thus, unnecessarily complex scheduling logic is removed from the processing core, so that the design implementation of the processing core is made as simple are possible. Advances in compiler technology thus result in improved performance without redesign of the hardware. In addition, some particular processing core implementations may prefer or require certain types of instructions to be executed only in specific pipeline slots or paths to reduce the overall complexity of a given device.

7

For example, in accordance with the embodiment illustrated in Fig. 2, since only processing path 56-1, and in particular execute unit 62-1, include a floating point processing unit 68, all floating point sub-instructions are dispatched through path 56-1. As discussed above, the compiler is responsible for handling such issue restrictions in this

5 embodiment.

In accordance with a one embodiment of the present invention, all of the sub-instructions 54 within a VLIW instruction word 52 issue in parallel. Should one of the sub-instructions 54 stall (i.e., not issue), for example due to an unavailable resource, the entire VLIW instruction word 52 stalls until the particular stalled sub-instruction 54

10 issues. By ensuring that all sub-instructions within a VLIW instruction word 52 issue simultaneously, the implementation logic is dramatically simplified.

3.   DATA TYPES

The registers within the processor chip are arranged in varying data types.

15 By having a variety of data types, different data formats can be held in a register. For example, there may be different data types associated with signed integer, unsigned integer, single-precision floating point, and double-precision floating point values. Additionally, a register may be subdivided or partitioned to hold a number of values in separate fields. These subdivided registers are operated upon by single instruction

20 multiple data (SIMD) instructions.

With reference to Fig. 3, some of the data types available for the compare sub-instructions are shown. In this embodiment, the registers are sixty-four bits wide. Some registers are not subdivided to hold multiple values, such as the signed 64 data type 300, unsigned 64 data type 304 and double precision floating point data type. In this

25 embodiment, the IEEE standards for floating point registers (e.g., ANSI/IEEE std 754-1985) are used for the floating point data types. However, the partitioned data types hold two thirty-two bit values within the sixty-four bit register. As those skilled in the art appreciate, there are other possible data types and this invention is not limited to those described above.

30 Although there are a number of different data types, a given sub-instruction 54 may only utilize a subset of these. For example, one embodiment of the byte swap sub-instruction may only utilize floating point data types. However, other embodiments could use different data types. The following discussion generally relates

8

to floating point compare operations, however, any data types could be compared in other embodiments of the present invention.

4.   COMPARE INSTRUCTION

Referring next to Fig. 4, the machine code for a floating point compare sub-instruction ("FPCMP") 400 is shown. This sub-instruction uses the register addressing form 400 where *Rs1* is the first input operand, *Rs2* is the second input operand and *Rd* is the output operand. The compare sub-instruction 400 is thirty-two bits wide such that a four-way VLIW processor with an one hundred and twenty-eight bit wide instruction word 52 can accommodate execution of four sub-instructions 400 at a time. The sub-instruction 400 is divided into an address and op code portions 404, 408. Generally, the address portion 404 contains the information needed to load and store the operators, and the op code portion 408 indicates which function to perform upon the operators.

The compare sub-instruction 400 uses the register addressing format to reference three general-purpose registers. First and second source addresses 412, 416 are used to load first and second source registers from the register file. A destination address 420 is used to indicate where to store the results into a destination register that could be any general-purpose register. Since each register 412, 416, 420 is addressed with six bits in this embodiment, sixty-four registers are possible in an on-chip register file 60. In this embodiment, all loads and stores are performed with the on-chip register file 60. However, other embodiments could allow addressing registers outside the processing core 12.

What is meant by the term "general-purpose registers" is that any number of different instructions can store their output operands in that register. For example, the third register 524-3 may hold the result from an addition instruction and later hold the result from a compare instruction. In this embodiment, some of the sixty-four registers may be dedicated to a special purpose, such as holding the program counter.

Bits 31-18 of the sub-instruction are the op codes 408 which are used by the processing core 12 to execute the sub-instruction 54. Decode logic interprets the op codes 408 to configure the processing core 12 to implement the desired instruction. Various sub-instruction types may have differing amounts of bits devoted to op codes 408.

The type field of the op code 408 located in bits 18-20 indicates which type of compare to perform. Table I lists several different *type* fields which correspond to different ways for performing a floating point compare sub-instruction. For a *type* of 000, the input and output operands are single precision. In *type* field 001, a single instruction multiple data instruction is performed on subdivided registers which each contain two single precision integer values. A type conversion is combined with a performing a compare function in *type* 010. The first input operand in *type* 010 is double precision and the second input operand is single precision. Performing a type conversion and compare with a single instruction increases the efficiency of the code. Finally, *type* field 011 performs a compare on two double precision integer operands and returns a double precision result.

Table I

Assembly Syntax	Type	Operand Description
fcmp.s Rs1, Rs2, Rd	000	Single precision input operands; Single precision output operand.
pfcomp.s Rs1, Rs2, Rd	001	SIMD instruction with two single precision fields in each register; Single precision input operands; Single precision output operands.
fcmp.ds Rs1, Rs2, Rd	010	Rs1 is double precision; Rs2 is single precision; Double precision output operand.
fcmp.d Rs1, Rs2, Rd	011	Double precision input operands; Double precision output operands.

Typically, a compiler is used to convert assembly language or a higher level language into machine code that contains the op codes. As is understood by those skilled in the art, the op codes control multiplexes, other combinatorial logic and registers to perform a predetermined function. Furthermore, those skilled in the art appreciate there could be many different ways to implement op codes.

5. COMPARE IMPLEMENTATION

The compare instruction determines the relationship between two register values. After execution of the instruction, an output operand is stored in the register file that can be read later to determine the relationship between the input operands. Knowing the relationship between the two input operands supports branches and conditionals that

10

are commonly used in the running of software. The output register stores an integer which indicates a relationship between the input operands of greater than, less than, equal to, or not a number (NAN) as shown in Table II. Subsequent branch instructions can read the output register to determine up to four different branch paths that depend upon the

5   value in the output register.

NAN values are defined in an IEEE standard (i.e., ANSI/IEEE Std. 754-1985) as invalid floating point values. The NAN value, which may be stored in the output register, indicates a comparison between the input operands is not possible. In other words, a NAN value means there is no valid relationship between the input

10 operands. For example, comparing two positive infinity values would return NAN to the destination register. Although the following discussion is of an embodiment that compares floating point values, any data types could be compared in other embodiments including, for example, integer types.

Table II

Relationship	Output Register
Rs1 is less than Rs2	-1
Rs1 is equal to Rs2	0
Rs1 is greater than Rs2	1
The relationship between Rs1 and Rs2 is indeterminate	Floating point compare not a number (NAN)

15

With reference to Fig. 5, a block diagram of one embodiment of the compare operation is shown. This embodiment 500 allows writing the result from the compare to any general-purpose register 524 in the register file 508. In other words, a dedicated compare output register with an associated path to that register are not required.

20 Included in the processing core 500 are decode logic 504, a register file 508, an arithmetic logic unit (ALU) 512, input operand load circuits 516, and an output operand store circuit 520.

The register file 508 holds the operands for an instruction. After execution of a particular instruction, the operands are available in the register file 508 for

25 subsequent instructions to use. For example, the result from a compare operation could be used to determine if a counter had decremented to zero which would indicate a

software loop were complete. Included in the register file 508 are about sixty-four general-purpose registers 524.

The machine code 400 for the compare instruction configures the processing core 500 to perform the compare operation. The op codes 408 and operand

5   register addresses 404 are interpreted by the decode logic 504 to configure the processing core 500 accordingly. The decode logic 504 uses the op code 408 to configure the ALU 512 for a compare operation. The *type* field of the op code 408 is used to select the data type of the operands, as discuss above.

The register address 404 portion of the instruction is used by the decode

10 logic 504 to select the proper registers 524 in the register file 508. The first input operand is read by the first load circuit 516-1 and the second input operand is read by the second load circuit 516-2. The load circuits 516 receive the input operand addresses 412, 416 from the decode logic 504 which are used to address the proper register 524 in the register file 508. Similarly, the output operand is written by the store circuit 520 to a register 524

15 in accordance with the output operand address 420 received from the decode logic 504.

The ALU 512 performs various operations upon the selected input registers to produce an output operand. For example, these operations include addition, subtraction and compare. The desired operation is provided to the ALU 512 by the decode logic 504 which determines the desired operation by analyzing the op code 408.

20 After performing the instruction, the ALU 512 passes the result to the store circuit 520 that writes to the appropriate general-purpose register 524 in the register file 508. As those skilled in the art appreciate, the operation performed in the ALU 512 could be pipelined to reduce the effective latency required for execution. Furthermore, the load and store functions could also be pipelined to further reduce latency.

25 Referring next to Fig. 6, a flow diagram is shown which depicts one embodiment of the steps for executing a compare instruction. In step 604, the instruction is decoded to determine which registers 524 to use for the operands and which function the ALU 512 should perform. The ALU 512 is configured in step 608 to execute a compare operation from the many different operations the ALU 512 could execute.

30 The address portion 404 of the instruction 400 is used to retrieve the register value 524 from the register file 508. In steps 612 and 620, the first and second input operand registers 524 are addressed such that a path from each addressed register 524 to the ALU 512 is created. Once the input registers 524 are properly addressed, the

12

input operands are loaded into the ALU 512 from their respective registers 524 in steps 616 and 624.

After the input operands are loaded, processing may begin. In step 628, the compare operation is performed in the ALU 512. As is well known in the art, a compare is implemented by subtracting the operands and analyzing the result. From this analysis, an value is formulated which indicates the result. The above Table II provides the integer values used in this embodiment to indicate the various results.

Once the result is formulated by the ALU 512, it is stored in the register file 508. The destination address 420 portion of the instruction 400 indicates which register 524 the result is stored to. In step 632, the address of the output register 524 is provided by the decode logic 504 to the store circuit 520. The store circuit 520 uses the address to create a path between the ALU 512 and the output register 524. Once the path is configured, the output operand is stored in the selected register in step 636. In this way, a compare instruction is performed.

Although not shown in Figs. 5 and 6, multiple compare sub-instructions could be executed in a single very long instruction word (VLIW). The use of general-purpose registers to store the output operands alleviates one problem with the conventional processing cores that use a dedicated comparison result register. In the present system, any number of general-purpose registers may be used to support simultaneously issued compare operations. Additionally, since there is no separate path to a dedicated compare register, circuit size and complexity is reduced.

Conclusion

In conclusion, the present invention provides a novel computer processor chip having a sub-instruction for comparing operands that allows performing several compares in a single issue, whereafter the result is written into a general-purpose register. While a detailed description of presently preferred embodiments of the invention is given above, various alternatives, modifications, and equivalents will be apparent to those skilled in the art. For example, although the operand processor only has a limited number of functions, other embodiments could include more functions. Therefore, the above description should not be taken as limiting the scope of the invention, which is defined by the appended claims.